# Python - Install and Setup - Mac OS X

A brief intro to installing and managing the Python programming language, including virtual environments for development, on Mac OS X.

## Contents

## Intro

There are various install and support options for the Python programming language depending upon your chosen operating system.

OS X currently supports a legacy default install of Python 2.x (e.g. 2.7.10 as of January 2017) for example.

However, Python 3.x is currently preferable for development unless there is a specific need to support legacy code from Python 2.x. There are various download and install options, again depending upon your chosen OS.

To correctly install and setup Python 3.x for use with virtual environments and Pygame, please read the following outline and instructions.

## OS X legacy

OS X already has a 2.x version installed,

```
$ python --version
Python 2.7.10 # as of January 2017
```

We'll now need to install and configure Python for version 3.x, and any required virtual environments.

## Check Xcode

Xcode is Apple's IDE (Integrated Development Environment), which comprises various tools for general software development on OS X. We can check the current state of Xcode with the following command,

```
$ xcode-select -p
/Applications/Xcode.app/Contents/Developer # standard return...
```

If this command returns an error, you'll need to install Xcode from the OS X App Store

Once Xcode is successfully installed, we may then need to install the separate *Command Line Tools* app, e.g.

```
xcode-select --install
```

### Homebrew

We may now use a package manager to install and maintain Python 3.x.

For OS X, we can use the Homebrew package manager. Further details on Homebrew install can be found at the following URL,

- GitHub - Homebrew

After successfully setting up Homebrew, we can check the current install version

```
$ brew -v # alternative command is `brew --version`
```

We'll also add the Homebrew directory to the top of the `PATH` environment variable. This helps ensure that Homebrew installations will then be called instead of various default tools OS X selects.

We'll need to update our `~/.bash_profile` file. Create a new one if it does not already exist,

```
$ touch ~/.bash_profile
```

Then edit with Nano &c.,

```
$ nano ~/.bash_profile
```

and add the following to this document,

```
export PATH=/usr/local/bin:$PATH
```

We can then save our updates, and activate them using the following terminal command,

```
$ source ~/.bash_profile
```

We can then check Homebrew is OK using the following command,

```
$ brew doctor
```

Now, we're ready to install Python 3.x.

### Install Python 3.x

We may simply start by searching the brew packages for current Python options,

```
$ brew search python
```

We'll then install the `python3` package,

```
$ brew install python3
```

This will install the latest version of Python 3.x, and associated tools, which include `pip`, `setuptools`, and `wheel`.

We can check the installed version as follows,

```
$ python3 --version
Python 3.6.0 # as of January 2017
```

We can then use Python's `pip` to install and manage any required packages for development. For example,

```
$ pip3 install package_name
```

`setuptools` helps with packaging Python projects, and the `wheel` package is a built-in package format for Python. This will often help speed up software production by reducing the need to repetitively compile code.

### Virtual environment

With Python, we can create programming environments with the tool `Pyenv`. This allows us to create any required virtual environments for project development.

A virtual environment allows us to isolate a project &c., thereby ensuring each project has its own set of dependencies. This helps ensure that each project remains isolated, and will not pollute or disrupt another project's development.

We can also use virtual environments to handle development of different versions, and easily handle required dependencies. This can be particularly useful for working with third-party packages.

It's often best to use a specific folder for adding such virtual environments, e.g.

```
$ mkdir dev-environments
$ cd dev-environments
```

Within this specific directory, we can then start to create our Python environments for application development and testing. e.g.

```
$ pyvenv test_env
```

or for Python 3.6 versions onwards, we can use the specific command,

```
$ python3.6 -m venv test_env
```

#### venv directory

Each virtual environment directory includes the following default files and sub-directories,

- `bin` sub-directory - includes a copy of the Python binary along with Pip, easy_install &c.
- `include` sub-directory - compiles required packages
- `lib` sub-directory - includes a copy of the Python version, e.g. `python3.6`, by default. It will then also save any third-party modules added to the project
- `pyvenv.cfg` - config file points to Python install used to generate virtual environment

These files and sub-directories help to isolate a given project from the broader context of a host machine. In effect, the virtual environment is isolated from the system files.

#### activate venv

To use each virtual environment, we'll need to activate it using the following type of command, e.g.

```
$ source test_env/bin/activate
```

This command is using the provided activate script, which will then prefix the system bash environment with the name of the new virtual environment, e.g.

```
(test_env) Your-MBPro:dev-environments username$
```

This prefix simply informs us that the virtual environment is now active. Any apps developed within this environment will be isolated, and use their own installed versions, packages, &c.

#### deactivate venv

We can then exit the current virtual environment using the following command,

```
$ deactivate
```

This will simply exit the current *venv*, and return us to the standard bash command line.

### Install Pygame

We can install Pygame using Homebrew and Pip, e.g.

#### `brew` dependencies

Check and install the following dependencies with Homebrew,

```
$ brew install mercurial
$ brew install git
$ brew install hg sdl sdl_image sdl_ttf
$ brew install sdl_mixer portmidi
```

#### `pip` install

```
$ pip3 install hg+http://bitbucket.org/pygame/pygame # if installed at default bash command line
$ pip install hg+http://bitbucket.org/pygame/pygame # if installed in virtual environment
```

This will then install `pygame-1.9.3.dev0` for the current system or virtual environment.

We can then simply test importing this module in Python, e.g.

```
$ python # if in virtual environment, otherwise use standard `python3.6` &c.
>>> import pygame
>>> pygame.init() #pygame will open, ready for display window &c.
>>> pygame.display.set_mode((800, 600)) # set and open display window for pygame...
>>> raise SystemExit # exit current pygame window...
```